Some Observations About Using SPIN and STeP to Verify StAC Specifications

Juan C. Augusto **Michael J. Butler**
{jca, mjb}@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2002-9
October 2002

www.dsse.ecs.soton.ac.uk/techreports

Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, United Kingdom

Some Observations About Using SPIN and STeP to Verify StAC Specifications

Juan C. Augusto Michael J. Butler
Declarative Systems and Software Engineering Research Group
Department of Electronics and Computer Science
University of Southampton
Southampton, United Kingdom

July 14, 2003

Abstract

Business transactions are prone to failure and to deal with unexpected situations. Some specification languages, e.g. StAC, introduce notions like *compensation* handling. Given the importance of verification of correctness in business related software, it is important to fill in the gap between specification languages like StAC and the verification software already available.

We report on two of our previous attempts to develop a tool to allow verification of StAC secifications by using already existing systems. One such attempts involved SPIN and its specifcation language, Promela. The other case involves STeP and its specification language, SPL. We focus on the problems we faced during these attempts, because they can prevent successful and widespread use of verification tools. Our experience can be used to make the available tools more versatile and hence, useful to a wider range of applications.

Contents

1 Introduction **4**

2 StAC **5**

3 Translating StAC to Promela **7**
3.1 Coordinating Nested Procedures 9
3.1.1 Sequential Invocations 10
3.1.2 Parallel Processes 11
3.1.3 Conditional Calls 12
3.1.4 Recursive Calls 12
3.2 Generalized Parallel 13
3.3 Early Termination 17
3.4 Handling of Compensations 19
3.5 Optional Labels 20

4 Translating StAC to SPL **21**
4.1 Recursive Specifications 24
4.2 Handling Enumerates 24
4.3 Flexibility on Generalized Operators 25
4.4 Early Termination 26
4.5 Obtaining Counterexamples 26
4.6 Proving Temporal Properties in STeP 27

5 Conclusions and Further Work **28**

A STeP Diagram **31**

B Counterexample for a Specification in STeP **32**

1 Introduction

Business transactions, because of their complexity, are prone to failure in many ways. For example, a request that normally is satisfied under certain conditions can be unexpectedly rejected. That can be experienced in daily life when the book we requested is not anymore in stock, when our trip is cancelled, or when an appointment we scheduled cannot be made effective.

However systems are normally built considering the normal and expected pattern of behavior from their many constituent parts. A way to deal with this conflict is to supplement that expected pattern of behavior with mechanisms that allows the system to react more appropriately when an unexpected/undesired event of this sort occurs. One such mechanisms proposed in the literature is the one of designing ways of *compensating* an action with a set of actions, that will repair or handle in some appropriate way that situation. Offering alternatives and rescheduling can be ways to compensate previous actions. A business process modelling language proposed to handle this concept is StAC (see [CGV+01] and [BF00]).

However the important issue of how to verify correctness in StAC specifications had not been addressed yet. We report here on our previous attempts to provide a suitable verification framework for StAC specifications. We considered two systems with different characteristics, SPIN [Hol97] and SteP [BBC+99]. The first option led us to consider a translation from the StAC specification language to Promela, the input language for SPIN. For the second option we considered instead a translatation to SPL, the input language for STeP.

We faced several problems preventing us from obtaining an easy to use verification framework by using any of these tools for the kind of bussines oriented specifications allowed in StAC. Hence we believe it is useful to provide a detailed explanation of the differences between these two classes of languages in order to guide future atttempts to verify business related systems.

We shall present our work beginning with a brief introduction to the main system involved, StAC (section 2). Following, we explain (section 3) the main obstacles preventing us from obtaining an effective translation from StAC to Promela. Then we give some details (section 4) about the problems we faced, both to achieve a translation from StAC to SPL and to use STeP at the verification stage. We finish this report with our conclusions (section 5) about how this experiences can lead to an improvement on next generation verification systems for business related systems.

2 StAC

StAC (Structured Activity Compensation) is a language that, in addition to CSP-like operators, [Hoa85] offers a set of operators to handle the notion of *compensation*. By this we mean a way to associate to an action a set of actions, typically one, providing a way to repair an undesired situation. Compensations are expressed as pairs with the form $P \div Q$, meaning that Q is the compensation planned in case that the effect of P needs to be compensated at a later stage. As the system evolves, possible compensations are remembered. Each compensation operator is bounded to a scope of application. If all the activities are successfully accomplished then the operator *accept* releases the compensations. If any activity fails, then the operator *reverse* orders the system to apply all the recorded compensations.

DEFINITION **1** Let A represent an activity, b a boolean condition, P and Q two generic processes, x a variable and X a set of values. Then, we can define as follows the set of well formed formulas in StAC:

Process	::=	A	(activity Label)
	|	0	(skip)
	|	$b \rightarrow P$	(condition)
	|	*rec(P)*	(recursion)
	|	$P; Q$	(sequence)
	|	$P \| \| Q$	(parallel)
	|	$\| \| x \in X.P_x$	(generalized parallel)
	|	$P [] Q$	(choice)
	|	$[] x \in X.P_x$	(generalized choice)
	|	$\odot$	(early termination)
	|	$\{P\}$	(termination scoping)
	|	$P \div Q$	(compensation pair)
	|	$\boxtimes$	(reverse)
	|	$\boxdot$	(accept)
	|	$[P]$	(compensation scoping) ▲

We now introduce two scenarios that will be useful to illustrate StAC and later on we will use as case studies when attempting to provide a verification framework for StAC. Those processes written in boldface are intended to be basic activities. Each StAC specification is coupled with a B machine [Abr96] describing the state of the system and its basic activities. Basically a B machine is composed by a declaration of sets, variables, invariants, initialization and operations over those structures. Each StAC activity in a specification will have associated an operation in the corresponding B machine explaining how that activity is implemented in logical terms, see for example [BF00] .

EXAMPLE **1** First we consider the specification of an order fulfillment scenario presented in [CGV+01]. The whole process can be described basically throughout the following steps: a) an order is accepted from a customer b) the warehouse prepare the order for shipment, including booking a courier for delivery c) simultaneously with step (b) there is a credit check to verify the costumer can pay the order. d) if the check is successful the order completes, otherwise it is stopped and the compensation mechanism is started. This can, for example, send a request for unpacking the ordered items. In this case we consider three items. We also include some variants to the specification given in [CGV+01] which results in a simpler translation process.

```
abc = (acceptOrder ÷ restockOrder);
      fulfillOrder;
      ((okFulfillOrder ; ☑ ) ▯(notokFulfillOrder ; ☒ ))

fulfillOrder = {wareHousePackaging ||
                  (creditCheck ;
                  ((notokCreditCheck;⊙ )▯(okCreditCheck;0))) }

wareHousePackaging = (bookCourier ÷ cancelCourier) || packOrder

packOrder = || i:{i1, i2, i3}.(packItem(i)÷ unpackItem(i))   △
```

EXAMPLE **2** Lets consider the specification of an E-Bookstore, presented in [BF00]. Here we consider a finite set C of registered customers. Each client is associated to a limited budget and also has an e-basket where to put the selected books. Every time the customer selects a book, its price is compared with the user's allowed budget. If the price exceeds the credit then it is returned to the e-shelf. When the customer asks to leave the E-Bookstore, either s/he pays or the selected books have to be returned to stock.

```
Bookstore = || c∈C . client(c)

client(c) = arrive(c); chooseBooks(c);
            ((quit(c) ; ☒ ) ▯ (pay(c); ☑ )); exit(c)

chooseBooks(c) = checkout(c) ▯ (chooseBook(c); chooseBooks(c))

chooseBook(c) = ▯b∈B . [(addBook(c,b) ÷ returnBook(c,b)) ;
                        overBudget(c) → ☒ ]

pay(c) = processCard(c) ; ¬ accepted(c) → ☒
```

`Arrive` creates and initializes the client information, that includes setting the appropriate budget. `ChooseBooks` represents the selection of books by the customer. Next, the customer is offered either to `quit` the transaction or to proceed by buying some items. In the first option the information is cleared just after the compensation is applied, i.e., after returning the books to the shelf, and in the second case it will be cleared after accepting the operation. ChooseBooks, specified as a recursive process, allows the customer to choose as many books as s/he wants. For each book the customer chooses, it is checked if that lead to exceed her/his budget. If it does, then the reverse operator causes the book to be returned to the shelf. `Pay` has the same effect in case that checking the credit card is not successful.

△

We address the reader who want a more in-detail account of StAC to [CGV+01] and [BF00]. The first article gives an explanation on how StAC relates to IBM's BPBeans enterprise technology and more examples. The second one also offers an operational semantics and the integration between StAC specifications and B Machines. Both explore how to consider the notion of *multiple compensation*, a feature we do not address in this work.

3 Translating StAC to Promela

Model checking [CES86] can be used to check whether a logical property holds for a finite-state system. Although extensions of this idea exists for the case of infinite state systems, we will focus on the finite case during this work. A particularly successful implementation of this approach is SPIN, [Hol97] that has been widely accepted as a tool to support the verification stages in software and hardware development.

It allows a series of useful verification actions in relation to a system specification with emphasis on efficiency. SPIN offers the possibility to perform simulations and verifications. Through these two modalities the verifier can detect absence of deadlocks and unexecutable code, to check correctness of system invariants, to find non-progress executions cycles and to verify correctness properties expressed in propositional linear temporal logic formulae. Promela is the specification language of SPIN. It is a C-like language enriched with a set of primitives allowing the creation and synchronization of processes, including the possibility to use both synchronous and asynchronous communication channels.

We refer the reader to the extensive literature about the subject as well

as the documentation of the system at Bell Labs web site [1] for more details, assuming some degree of familiarity with this framework from now on.

Translating StAC specifications to Promela proved to be a non trivial matter and, when possible, demanded much more complex data and control structures to recreate StAC novel features in Promela syntax. At some level of abstraction and for some operators we can find a relation between some operators in StAC and some others in Promela. For example, regarding control structures, we can find the following relations.

- *sequence*, represented in stack with the symbol "**;**", can be modelled with the same symbol in Promela in the case of basic activities.

- *choice* of type `((`*condition1* `;` *Pr1*`)` $[]$`(`*condition2* `;` *Pr2*`))` can be dealt with by using an `if ... fi` structure in Promela.

- *parallel* operator, represented in StAC as "$\|$", can be implemented in Promela by running two processes. For example, *proc*1 $\|$ *proc*2 can be represented as `run proc1() ; run proc2()`.

A key notion in any non-trivial StAC specification is that one of a non-primitive process. Basically they are used in the same way as procedures in high level programming languages. There is no direct way to define procedures in Promela. However, they can be implemented supplementing `proctype` declarations with some coordination mechanisms to ensure their behavior is synchronized in the same way as expected in StAC. For example, as invoking processes in Promela means their execution is started then starting sequential processes means that processes can be interleaved instead of run strictly in sequence. Similar problems can appear in the context of StAC specifications while using parallel, conditional and recursive operators. We consider each of these cases with more detail in section 3.1.

- *generalized parallel*, $||i \in S.P_i$, demands more work as Promela does not have a constructor that allows us to directly implement this concept. However it can be built by combining different concepts already at hand, i.e. proctypes and some mechanism for synchronization. This will be explained with more detail in section 3.2.

- Other non-StAC related operators like *skip* and *condition* do not pose much interest from the perspective of this article in the sense they can be translated straightforwardly to a Promela syntax equivalent.

[1] `http://netlib.bell-labs.com/netlib/spin/whatispin.html`

Those operators more specifically related to the compensation notion have to be handled with special purpose procedures resorting to different Promela resources.

The *early termination* operator is applied within a *termination scope* and allows to terminate all processes in that scope. However, implementing that in Promela can be very cumbersome as it does not provide an equivalent concept, especially when we need to terminate parallel processes. We comment on that problem in section 3.3.

A *compensation pair* demands remembering what compensation operation has to be applied associated with a particular action. Because compensations are applied in reverse order, [CGV+01] pp. 4, it can be implemented putting into a stack the code identifying the compensation to be remembered. Given the above mentioned implementation of compensations using a stack the *Accept* compensation will pop up elements out the stack forgetting the compensation while the *reverse* operator will pop the code from the stack and call the corresponding compensation action accordingly. However, the complexity of the information to be stored can turn this into a non trivial matter, as it will be explained in section 3.4.

It is desirable to have a standard and easy way to refer to special places or processes in the specification to be verified. One way to do that is to introduce labels during the translation process into the resulting code in such a way they can be referred in an easy way inside temporal logic formulas. We explain in section 3.5 why this is problematic to be implemented in Promela.

3.1 Coordinating Nested Procedures

Calls to non primitive processes in StAC behave as calls to procedures in programming languages. For example, a sequence of calls to non-primitive processes in the StAC specification must be executed without interleaving between them, while their proctype counterparts in Promela will allow interleaving. For example, `run P; run Q` will start `P` first and then will start `Q` without waiting for `P` to terminate. `Q` can be started at any time after `P` has been. Then the `;` operator in this case does not have the usual semantics for procedures in high level programming languages as it is the case for StAC.

Synchronization can be achieved as expected in StAC trough a fork & join mechanism forcing all subprocesses to be finished before the process that created them is considered finished. Broadly speaking, a way to introduce this mechanism in the translation is as follows, for every process P calling other subprocesses $p_1, \ldots, p_n$, i.e. implemented as `proctype` calls through the `run` sentence, we can a) add at the end of each p_i, with $i = 1, \ldots n$, a way to acknowledge that the process finished and b) after the call sentence

in the caller process a way to recognize that the called processes finished before proceeding. This is just a simplification to present the general strategy because there are further subtleties to consider depending on the context where the call is performed. In addition to sequence and parallel, other structures affecting coordination of processes invocation are conditionals and recursive definitions. We devote this section to explain how to deal with these four options.

Note that the alternative solution to use *inline* definitions does not work in the case of recursive definitions, something fairly common in StAC specifications.

3.1.1 Sequential Invocations

for a StAC specification: `... A; B ...` we want `A` to be finished before starting `B` at all. To achieve that we supplement `A` and `B` with acknowledgments at the end of their definitions and the calls to `A` and `B` with tests to check they have finished. In this way first `A` will be called and only after its acknowledge is detected `B` will be invoked.

EXAMPLE **3** A sequence `... A; B ...` in our framework is translated into Promela as the following supplemented invocations:

```
    chan ch_A2caller = [0] of {bit}
    chan ch_B2caller = [0] of {bit}

proctype A() { ... (main body of A) ...; ch_A2caller!1 }
proctype B() { ... (main body of B) ...; ch_B2caller!1 }

proctype caller()
{ bit doneA=0, doneB=0;

  run A();
  ch_A2caller?doneA;
  do
  ::  doneA ==1 -> break
  od;
  run B();
  ch_B2caller?doneB;
  do
  ::  doneB==1 -> break
  od
}
```

△

It is worth to notice that: (a) this mechanism is independent of the amount of callers a process has and (b) it is also independent of how many simultaneous calls a process have because each call from different processes will generate a dedicated channel to handle their coordination, e.g. like `ch_A2caller` and `ch_B2caller` in the previous example.

To simplify further explanations on this issue lets assume the following notational conventions. We keep using the term *caller* to identify a process that call others and we introduce the term *called* for any process that is called, e.g. `A` and `B` in example 3. We call `acknowledge` the message sent by a called process to a caller, e.g. ch_B2caller!1 in example 3. We represent by `test_`*(Name of Proctype)*`_finished` both the channel request and the `do ... od` loop waiting for the acknowledge after each call, like those after `run A()` and `run B()` in example 3. By using this terminology we will sketch how to proceed in other similar cases, namely calls made in the context of parallel, conditionals and recursive structures.

3.1.2 Parallel Processes

A parallel operator can be used to invoke processes in many ways. Perhaps the situation of two invocations like in `A(...) || B(...)` is conceptually the easiest to think about. However, it could also happen even when some or none of the processes involved are actually invocations e.g. `A(...) || Pr` and `Pr1 || Pr2` where `Pr, Pr1, Pr2` can be any legal process in StAC, different from an invocation to a process. If any of the processes involved in the use of a parallel operator is not an invocation, then has to be delimited in some way, e.g., `A(...) || (Pr1; Pr2)` tell us that in parallel with the invocation to `A`, a composite processes made up of a sequence of processes `Pr1` and `Pr2` is executed. We will call the parallel definition to be coordinated *a block*. A block `A || B` terminates when both `A` and `B` terminates. As a difference with the sequential case we want to ensure now that parallelism is implemented but just restricted to those processes in the block. In this case the testing for acknowledgments is shifted immediately after the translation of the intervening processes inside the block.

Example **4** If we consider the case of coordinating a block like `A(...) || B(...)` we include "`chan ch_A2Proc = [0] of bit`" as a declaration and generate the following code in a second phase of the translation:

```
proctype A() {(body of A) ... ch_A2caller!1 }
proctype B() {(body of B) ... ch_B2caller!1 }

proctype caller()
```

```
{ bit doneA=0;
  bit doneB=0;

  { /* begin block */
    run A(...); run B(...)
  } /* end block */

 test_A_finished;
 test_B_finished
}
```
△

3.1.3 Conditional Calls

Calls to a set of procedures can be performed also from "branching" structures, e.g., conditions and choice operators. In such cases, different tests for termination must be attached to each branch.

EXAMPLE **5** For example, the following specification in StAC includes the use of a choice operator: `(a=0; (A; B)) [] (a>0; C)` where `A, B,C` are invocations to non-primitive processes. It can be mapped into the following Promela code.

```
  if
  ::  a=0 -> run A(); test_A_finished;
             run B(); test_B_finished
  ::  a>0 -> run C(); test_C_finished
  fi
```
△

3.1.4 Recursive Calls

StAC allows recursive definitions (see definition 1). We can adapt the idea used in the previous case for conditioned calls to recursive specifications. A recursive definition can be seen as a special case of choice, offering at each step either the opportunity to use the termination case or the recursive case. Termination cases must acknowledge they finished successfully (here that part of the declaration is acting as a `called` process) while in the recursive case a test must be added (having the role of a `caller` in our initial generic explanation).

EXAMPLE **6** Lets consider for example the following StAC specification of a recursive process `A(n)`:

```
(a=0; finish=true) [] (a>0; a=a-1; A(a))
```

It can be mapped into the following Promela code:

```
proctype A(short a)
{
  if
  ::  a=0 -> finish=true; acknowledge
  ::  a>0 -> a=a-1; run A(a); test_A_finished
  fi
}
```

△

3.2 Generalized Parallel

Either if we use a parallel or a generalized parallel, operatorPAR_i for short from now on, we want to ensure that we consider the process finished if and only if all the processes being run concurrently are finished. Then again we need to address the coordination problem, we focus in the most general case, the PAR_i operator, because in addition to the coordination problem it involves issues of legibility when we want to consider a translation. The number of calls a generalized operator can generate is usually higher than those made by individual calls scattered throughout the code. Then, using the same schema than in section 3.1 can result in an uncomfortable writing for testing termination of called processes.

EXAMPLE **7** If we translate `|| i: {i1, i2, ..., in}. A(i)÷C(i)`, for a particular n (we leave that generically specified to emphasize that could be any finite number), that would lead to the following translation schema:

```
ch_A(1)2caller?doneA(1);

...

ch_A(n)2caller?doneA(n);
do
::  (doneA(1) ==1 && ... && doneA(n) ==1) -> break
::  else      -> skip
od
```

△

Becasue number n can be arbitrarily large, then the conditional `(doneA(1) ==1 && ... && doneA(n) ==1)` and a long list of receive channel operations can make the resulting Promela code hard to read. Then, the

approach that we followed to coordinate common procedures and was useful in that context is not appropriate anymore. Because we are also interested on producing a Promela specification that can be directly readable as an alternative specification for the system under development then we consider an alternative way to enforce coordination in this cases.

We can generalize the previous approach as follows. We can keep the acknowledgement procedure for called procedures but to make a more general, and still compact, testing mechanism we can adopt a buffered channel of an appropriate size to store as many messages as processes will be created. Again what the "appropriate size" is can be collected during the first of our two-phase translation. In this case the condition of termination is tested using the `full` Promela sentence to detect when all processes sent their acknowledges, i.e., when the buffer has been filled of acknowledgements messages.

EXAMPLE **8** Lets assume `A` is the called process, n is the amount of processes called by the generalized parallel operator, and a declaration

```
chan ch_A2caller = [n] of bit
```

has been created after the first phase of the translation. Then, the translation of the PAR_i operator considered in the previous example will be instead:

```
do
::  ch_A2caller?index_ch_A2caller; index_ch_A2caller++
    /* fill the queue */
::  full(ch_A2caller) -> break
    /* done           */
od
```

△

Observe this problem does not apply to *generalized choice* because if the choice involves a procedure call all we need to check is that one call was made and for that purpose we follow the former strategy.

However, a problem that applies to both, *generalized parallel* and *generalized choice*, is that the above described schema assumes that values to be used with generalized operator are numeric. But, the usual case is to provide different enumerates for each operator representing names, brands, addresses and all sort of useful labels motivated by real life applications. So, if a more flexible set of values has to be used, the limitations to use enumerates in Promela are evident. Although a special range of constant values can be defined as an enumerate definition using `mtype` the documentation warns the

reader that just one of this definitions can be used on each Promela specification. If all values from all enumerates are put together the practical use is very limited because there is no implicit type checking performed for each subtype. Although it is still not implemented a feasible alternative solution, expensive both in terms of legibility and performance, is to build an enumerate type upon the `mtype` and the possibility to define new types in Promela. The following sketch of code illustrates the idea applied to example 2. We provide constant and variable declarations, then we use the structure in the translation of parameterized parallel and choice, and finally we provide the initialization of the structure in the main process. There are two instances of the more problematic cases of proctypes calls to exemplify the use of the new proposed structure. One is calling a generalized parallel as when we neeed to call `client` inside `bookstore` and the second is an example of calling a generalized choice like when we call `chooseBook`.

```
/* Constant declarations */

/* how many elements the enumerate type has?
   (to be calculated during translation time) */
# define max_enumerate

/* Type declarations */

mtype = {john, mel, java, prolog, xml}

typedef E
{
mtype el[max_enumerate]; /* possible elements     */

short types[max_enumerate];  /* possible types        */

bit   current_type           /* index to current type */
}


/* Variable declarations */

E e %  (e is a variable of type E)

/* "enumerate" will have for example the following settings:
```

```
el           : John mel java prolog  xml ...

types        : 0    0   1    1       1 ...

current_type : 0

to express that the valid enumerate for this call is (john, mel)*/
```

To implement generalized parallel or generalized choice, the `proctype` that is substituting the generalized parallel or generalized choice operator has to be invoked with e.type instantiated with a number to select the corresponding type. For example, if the following `proctype` definitions are intended to replace, respectively, `Bookstore = || c∈C . client(c)` and `chooseBook(c) = []b∈B . ...` in the ebook case study:

```
proctype callClients(E e)
{

/* in this case we just care about the type of element as the
proceedure will be the same for all those elements */
       ...
       index=0;
       do
       :: (index < max_enumerate
           &&
           e.types[index] == e.current_type)
                             -> run client(e.el[index]);
                                index++
       :: (index < max_enumerate
           &&
           e.types[index] !== e.current_type) -> index++
       ::  else                               -> break
           /* index >= max_enumerate */
       od;
       ...
}

/* to implement generalized choice. */

proctype chooseBook(mtype book)
{
```

```
/* in this case we need to apply ad-hoc procedures for each
element */

   %  Generalized Choice
      if
      :: book=java
      :: book=prolog
      :: book=xml
      :: else --> printf("error in book type")
      if;
      ... (proceed accordingly)
}
```

Then the following initialization can be created by the translator to allow the right interpretation of the enumerate on each call:

```
init{e.el[0]=john; e.types[0]=0;
     e.el[1]=mel; e.types[1]=0;
     e.el[2]=java; e.types[2]=1;
     e.el[3]=prolog; e.types[3]=1;
     e.el[4]=xml; e.types[4]=1;

     e.current_type = 0;
     run callClients(e);
     run chooseBook(e)
}
```

It can be seen that, altough possible, it demands an involved procedure to overcome the restrictions imposed in Promela to the use of enumerates.

3.3 Early Termination

The *early termination* operator, $\odot$ (see example 1 for an illustration of its use), can be applied to force a process to terminate. Brackets can be used to delimitate the scope for the operator application. For example $\{P; \odot ; Q\}; R$ specifies that after P is executed, Q will be forced to terminate. This will not affect R. If we apply $\odot$ to a parallel process then all the parallel processes within the scope of the $\odot$ are also terminated. For example, in $\{(P; \odot ; Q)||R\}||S$ process R will also be terminated but S will not. We found that the implementation of this characteristic is particularly problematic in Promela.

The closest approach to a solution being the use of the label `provided` available in Promela which impose conditions to the executability of a proctype, provided some conditions are fulfilled. Then, the obvious move is to make dependant the parallel processes on a condition that is falsified as soon as the early termination operator is applied. A sketch of such a specification follows:

```
bool early_termination_caller

proctype called() provided (early_termination_caller == false)

{ ...(called code)... }


proctype caller() {
   early_termination_caller = false;
   run called();
   if
   :: (condition1) -> early_termination_caller = true;
                      goto label_early_termination_caller
   :: (condition2) -> ... (code to be used under normal conditions) ...
   fi;

label_early_termination_caller: skip

/* it must be the entry point to the acknowledge section */
          ...
}

init {run caller()}
```

Conceptually speaking this way to solve the problem is correct but a note in Promela users' manual warns the user: `" ... provided clauses are incompatible with partial order reduction ... "`. This make the solution inconvenient from a practical perspective as partial order reduction plays an important role in SPIN efficiency and even in the possibility to solve a problem or not.

3.4 Handling of Compensations

A FIFO structure is used to record compensations, as a result, when the compensation mechanism is applied they will also be compensated following that strategy. As the stack used to implement the notion of compensation is made up of global structures, access to these structures should be coordinated amongst the various concurrent processes. This prevents other concurrent processes from attempting to access that shared resource. In Promela there is a simple and efficient way to forbid interleaving over a set of statements by using either an *atomic*{...} or a *d_step*{...} (deterministic step) statement. The last option although more restrictive on their applications is anyway suitable for our simple sequence of assignments as well as the more efficient in terms of verification. More details about the differences between *atomic* and *d_step* statements can be found in the online Promela grammar definition.

EXAMPLE **9** Lets assume `P ÷ C` is the compensation to be translated and `c` is a code assigned during the translation process to represent compensation `C`. Then, the resulting translation is just the storing of the compensation code `c` into the stack:

```
d_step{index++;
       compensations[index]=c
      }
```

△

Stored codes can be recovered later, if necessary, to know what compensations must be applied and in which order that must be done. Then each compensation must have an identification and all must be unequivocally identifiable, i.e. the function relating codes with compensations is bijective. Otherwise, recovering a code from the compensation stack could lead to the application of the wrong compensation. To ensure that each compensation has a different code we need to force each generalized parallel affecting a compensation pair to generate a disjoint set of codes with respect to those used in other applications of compensation pairs. But we cannot know in advance what codes will be required in other parts of the specification.

This could be solved by using a two-phase strategy to perform the translation from StAC to Promela, postponing the assignment of codes for generalized parallel operators at the end of the translation. During the first step, information is gathered about PAR_i operators. In the second stage, the actual writing of the output, we are able to generate the constants, variables and loops specifically tuned to simulate the different calls that a PAR_i operator can produce at run time. In this way we neatly divide the problem in two

clearly identifiable stages: a) find out how many different codes are needed from individual applications of compensation pairs and how many different codes each compensation pair affected by PAR_i operators will generate b) during the writing phase attach different codes for any use of compensations affected by PAR_i operators.

3.5 Optional Labels

As a part of our research we considered the possibility to generate automatically some set of labels that help to identify clearly when some key states of the system are holding. This labels are not introduced compulsory in the translation as there is a tradeoff between the amount of variables in the Promela specification and the performance of SPIN during the verification process. Instead, they are offered as options to the user, being allowed to generate different specialized Promela translations depending on what set of properties is s/he interested in.

One type of labels that can be set during the verification process are those we call "location labels". They are intended to provide a clear indication about different key places in relation with the processes involved in the specification. For example, in the order fulfillment scenario given in example 1 we may want to check that whenever the `fulfillOrder` process is invoked, it will finish. Two location labels can be associated with this aim checking that always reaching the beginning of `fulfillOrder` will lead to reaching the last line of its specification. If there is no clear way to indicate the process has reached this point then some equivalent code lines have to be identified with that situation and their use can significantly obscure writing the temporal formulas for verification.

One tempting approach to explore in order to prevent state explosion could be by using labels provided in Promela to identify meaningful parts of the specification. However, there seems to be no way to use these Promela labels in a useful way at the verification stage. For example, the *remote reference* mechanism to refer to the special location `L` in the only instance of process `main` by `main[0]@L` is not allowed as a valid proposition in the LTL syntax.

Sometimes it is also useful to have an easy and unambiguous way to state if during a computation a given process has been started, is being executed or has finished. The mechanism we use provides an alternative way to "label" locations, or more generally situations. It can be implemented with a pair of boolean variables. For a process `P` we can assume two boolean variables `beginP` and `endP` that are assumed to be initially false. As the first sentence in process `P` it can be inserted: `d_step{beginP=true; endP=false}`. Anal-

ogously, at the end of process `P` it can be inserted: `d_step{beginP=false; endP=true}`. Then the relation of a process in relation to a computation can be tracked throughout the values of this two location variables: $(false, false)$ holds when the process has not been activated yet, $(true, false)$ when it has been activated and is still active, and $(false, true)$ when the process finished one of its activations. Referring to repeated activations demand to use at least two of these pairs. For example, to express that the process can be reinvoked it is necessary to say that after $(true, false)$ holds, $(true, false)$ can hold at some future computation.

More interestingly, "compensation labels", again boolean variables appropriately set, can be added after compensation-related operations. Because this can be done during translation time and the name of both the action to be compensated and the compensation itself are known, meaningful names can be chosen for those variables. In example 2 after generating the code for compensation `ReturnBook` an assignment `rememberedReturnBook=true` can be set. We can also use this mechanism to highlight the application of relevant operators like `reverse` and `accept`, setting boolean variables like `rememberedReturnBook` to false when any of this operators is applied. Then as part of an investigation to study the proper behavior of reverse we can built formulae like:

$$rememberedReturnBook \wedge reverse \rightarrow \Diamond \neg rememberedReturnBook$$

As we explained before this possibility to post labels must be used carefully because although the translation time is dismissable, the space search during model checking can be incremented considerably depending on the amount of labels posted by the user.

4 Translating StAC to SPL

STeP ([BBC+99]) is a verification system for reactive systems based in a deductive approach. The aim of the system is that allowing verification in a deductive framework allows infinite-state systems to be considered, as opposed to the model-checking technique that only allows finite-state system verification. STeP provides a collection of tools allowing verification by deduction, sometimes with user interaction. It provides verification rules and verification diagrams, automated support for proving verification conditions. Other more specific features related to the deductive framework are: automatic generation of invariants at different levels of complexity, simplification of equations and conditions, decision procedures and first-order reasoning. Model checking is also available and is a good complement to the deductive

system providing counter examples to false properties. In this case, model checking is provided in both forms: explicit-state and symbolic. There are extensions for Real-Time and Hybrid Systems. See Appendix A for a general diagram of STeP.

A system can be input to STeP as an SPL program or as a *Fair Transition System* [MP92]. Systems specified in either of these notations must be given to STeP as files. A specification file has three parts:

1. declarations (types, values, macros, rewrite rules, and auxiliary variables)

2. axioms (properties that can be assumed by STeP in a verification)

3. properties (the formulas we want to prove true for a program or system)

The syntax of SPL programs follows that of traditional imperative languages such as Pascal. In addition to the basic constructs found in these languages, SPL supports nondeterminism by means of the selection statement '`or`' and parallel composition by means of the cooperation statement ||. Parallel processes can interact through shared variables such as semaphores, as well as by synchronous and asynchronous channels. Execution of parallel processes is assumed to proceed by interleaving. An example of a specification as an SPL program for Euclid's algorithm is given next:

```
in a,b: int where a > 0, b > 0
local x,y: int where x = a, y = b
out gcd: int l0:
   [loop forever do
      [ t1: guard x > y do x := x - y
      or
        t2: guard x < y do y := y - x
      or
        t3: guard x = y do gcd := x
      ]
    ]
```

An equivalent specification as a transition system follows:

```
Transition System % Euclid's algorithm
in a,b: int where a > 0 /\ b > 0
local x,y: int
out gcd: int
Initially x = a /\ y = b
Transition t1 Just:
    enable x > y
    assign x:= x - y
Transition t2 Just:
    enable x < y
    assign y := y - x
Transition t3 Just:
    enable x = y
    assign gcd := x
```

For the sake of space we address the reader interested in specifying systems using SPL syntax or Fair Transition System notation to [MtSg95], chapter 2. The specification language for temporal properties to be checked is Linear-Time Temporal Logic [MP92].

More documentation about the system, including tutorials, demos for specific parts of the system and case studies can be found in the web page for STeP [2]. Some case studies available on-line are: A Ring Based Leader-Election Algorithm, Model Checking the Needham-Schroeder Protocol, A Round-Robin Bakery Algorithm (all by Calogero Zarba), a Verification of a DLX Pipeline (by STeP team), A Demarcation Protocol (by Jeff Kamerer), A split Transaction Bus (by Jeff Kamerer), and The Ricard-Agrawala Protocol (by Jeff Kamerer).

Now we focus on some of the obstacles we came across while repeating our previous attempt to achieve automatic translations for verification of StAC specifications. This time the target language being SPL. We begin enumerating some limitations we found in SPL to represent all the concepts required in StAC, e.g. using recursive specifications (section 4.1), handling enumerates (section 4.2), using generalized parallel or generalized choice (section 4.3) and implementing the $\odot$ operator (section 4.4). Later on we consider some undesirable features we found while using some of the tools provided in STeP. In section 4.5 we comment on some problems we found using the model checker.

[2] http://www-step.stanford.edu/

4.1 Recursive Specifications

Because "the parser just plugs in the bodies of procedures when it finds a procedure call" ([MtSg95], pp. 29), recursion cannot be used as needed in StAC. To overcome that we have to resort to an equivalent translation, e.g. a While-like loop. Lets consider the specification of `chooseBooks`, a process allowing a customer to choose 0, 1, 2 or more books:

```
chooseBooks(c) = checkout(c) [] (chooseBook(c); chooseBooks(c))
```

It cannot be directly translated to SPL as a recursive procedure but instead it could be written in an equivalent way as an iterative procedure. Following we sketch a possible solution that works for tail-recursive specifications. As a part of the translation we introduce a boolean variable `act_checkout` that is assigned values nondeterministically (first sentence) representing the possible choice of any customer to leave the process, even without choosing a book:

```
[act_checkout := true or act_checkout := false];
while (∼ act_checkout) do
      [ chooseBook(c);
        [act_checkout := true or act_checkout := false]
      ]
```

4.2 Handling Enumerates

SPL does not allow repetition of enumerate values and that can be problematic to check types for input data in our specifications. For example, lets consider that in our *ebook* case study we sometimes need to refer to different sets of books under different considerations, say by subject or by alphabetic order. Then we may need different types:

```
type Subject_CS       = {java, prolog, xml}
type Subject_Fiction  = {...}
type Alpha_Order      = {..., java, ..., prolog, ..., xml, ...}
```

but in SPL we are forbidden to specify `Subject_CS` and `Alpha_Order` because they have some elements in common. Should we need to create an enumerate with a combination of previous enumerate definitions then we need to resort to conventions or mappings to overcome the limitation. For example, one possibility in SPL is to encode enumerates numerically and treat them consistently throughout the specification. Below we use subrange declarations to replace the enumerate subtypes:

```
type Subject_CS       = {1..3}
                        % 1 = 'java'
                        % 2 = 'prolog'
                        % 3 = 'xml'
type Subject_Fiction  = {4..7}
type Alpha_Order      = {1..7}
```

Although uncomfortable still this make feasible to deal with StAC specifications using variables, but as a result

- we need to consider this issues during the translation introducing case-like sentences to map strings to numerical codes and viceversa. For example, when a compensation has to be applied or when feedback has to be given to the user.
- the resulting SPL translation is more difficult to read and modify.

4.3 Flexibility on Generalized Operators

SPL do not allow to use non-numeric enumerate values in generalized choice and parallel. For example, we cannot use:

```
[or O[c:[java..xml]] ::   ...
```

Again we have to resort to encodings, mapping strings into numbers and use the numbers as a metaphor of the real information with the same negative consequences than in the previous step.

Also, although is not clear form the documentation why, using generalized choice in SPL seems to be computationally expensive. For example in the ebook case study we would like to use a compact specification like:

```
or O[c:[1..2], b:[4..6]] ::  [proc_addBook(c,b)];
```

but changing that specification by the equivalent

```
[crandom := 1 or crandom := 2];
[brandom := 4 or brandom := 5 or brandom := 6];
proc_addBook(crandom,brandom);
```

made the difference between the model checker being not or being able to finish the diagnosis.

4.4 Early Termination

SPL does not provide any constructor that can help to implement the *early termination* operator. There is nothing in SPL syntax similar to the label `provided` available in Promela to impose conditions on the executability of a process.

This force us to implement that notion by using especial structures and procedures to represent that notion when using STeP. Means to detect when a concurrent process has finished must be implemented by using a structure where to store inter-related processes and conditions of termination have to be inserted inside inter-related processes to make their executions threads dependent on each others computations.

4.5 Obtaining Counterexamples

Interpreting a counterexample given by the model checker is a very involved process as steps provoking the unexpected situation are explained in terms of internal variables acting as indirect references to the user's structures. To illustrate this we consider a small specification involving a permanent loop. We just have one procedure without parameters, `p`, when the procedure is called the conditions for going out of the while loop are never met:

```
local b: bool where b = false
local end: bool where end = false

procedure p()
[ b :=true;
  while b do b:=true;
  end := true
]

p()
```

So asking if the location where variable `end` is assigned `true` could ever be reached will produce the model checker to give a counterexample as answer. The counterexample provided by STeP is included in Appendix B.

There seems to be no syntax explanation in any of the publicly available documentation for the system. This force users to have a deep knowledge of all the theoretical framework underlying the proposal in order to be able to understand a counterexample.

4.6 Proving Temporal Properties in STeP

After finding a way to map the characteristics of StAC specification language to SPL we were able to prove temporal properties characterizing the behaviour of the original StAC specification. We list below some safety and liveness properties we investigated for the ebook case study:

```
% Safety Properties ----------------------------------------
% (1 is the encoding for "John" and 2 for "Mel")
% (4 is the encoding for "Prolog", 5 for "Java" and 6 "XML")

% it is true that
% "No decision is made until the system is certain
% about a successful transaction"
[](!((accept[1] \/ reverse[1]) /\ (<> act_processCard[1])))

% it is true that
% "The system cannot proceed with the credit card
% transaction and still being allowed to choose books"
 [](!(!(act_checkout[1])  /\ act_processCard[1]))

% it is false that
% "It is not possible to deal with more than one costumer
% at a time"
 [](!( act_arrive[1] /\ act_arrive[2]))

% it is true that
% "at least one authorized customer is allowed to operate"
 <>(act_arrive[1])

% it is false that
% "no unauthorized customer are allowed to operate"
 <>(act_arrive[3])

% Liveness properties -------------------------------------
% Notice: '[] (A --> B)' becomes 'A ==> B' in STeP notation

% it is true that
% "Each customer is forced to leave the system either by
% releasing all items on her/his basket or by paying"
  act_checkout[1] ==> (<> (act_quit[1] \/ enteredPay[1]))
```

```
% it is true that
% "If  the transaction was not successful at some point the
% acceptance operator should not be applied at any time"
 ((<> ~okprocessCard[2]) ==> ([] ~accept[2]))

% it is true that
% "No books are returned except in case of exceeding the
% allowed budget"
 [] (!(e=(1,4) \/ e=(1,5) \/ e=(1,6))) Awaits (act_overBudget[1])

% it is false that
% "Each customer is forced to do a transaction"
 act_arrive[1] ==> (<> act_processCard[1])
```

5 Conclusions and Further Work

We focused on the translation process from StAC specifications to Promela and SPL the input languages for SPIN and STeP, respectively. For the first attempt, using SPIN, we found many problems to map a high level language as StAC to the structures provided in Promela. A translation is feasible and for specifications which does not involve parameters and the early termination, the translation can be used to do some basic verification. That may include temporal formulas depending on the length of the specification considered. However, for case studies like the `ebook`, and possibly many or the majority of real StAC specifications, parameters are needed and then complexity of translation and space exploration of the resulting model check process will increase up to an undesirable level.

SPL is more similar to StAC in structure so we faced less problems trying to match both languages. In the declaration section, even when enumerates are not so flexible as we may expect, we can write different enumerates. About the general structure of the specificatoin we also have some improvements because procedures are handled in much the way we intend them to be in StAC so we do not need to think about synchronization issues and that make the resulting code clearer and shorter. Some StAC constructs are mapped quite straightforwardly into equivalent SPL sentences. Probably the two more important examples of this are the generalized parallel and generalized choice operators. Still we are faced with some problems to handle specifications with variables due to restrictions imposed over data types. In this case many problems arise from the lack of documentation and support available for the system. It is also problematic to interpret counterexam-

ples in STeP. In order to make easier the comparison between both SPIN and STeP we focused in the model checking facilities of STeP although it has to be said STeP is a deduction-based verification environment and the model checking facilities are provided mainly as a complement to the theorem prover. Still the translation procedure from StAC to SPL need more research and has not been actually implemented so there could be more drawbacks to discover during that stage.

Although none of the tools were completely satisfactory for our goals, we found SPIN more satisfactory as the environment to work with while the specification language for STeP was closer to our needs.

Acknowledgments

We would like to thanks to Carla Ferreira for her contributions in relation with StAC specifications.

References

[Abr96] J. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University, 1996.

[BBC+99] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, B. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Formal Methods in System Design*, 16:227–270, 1999.

[BF00] M. Butler and C. Ferreira. A process compensation language. In *IFM'2000 - Integrated Formal Methods, volume 1945 of LNCS*, pages 61–76. Springer Verlag, 2000.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finitestate concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGV+01] M. Chessell, C. Griffin, D. Vines, M. Butler, C. Ferreira, and P. Henderson. Extending the concept of transaction compensation. Technical Report. Declarative Systems and Software Engineering Research Group, Dept. of Electronics and Computer SCience, University of Southampton. To appear in IBM Journal of Systems and Development, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol97] Gerard Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems (Specification)*. Springer Verlag, 1992.

[MtSg95] Zohar Manna and the STeP group. STeP: The Stanford Temporal Prover (Educational Release), User's Manual. Technical report, 1995. STAN-CS-TR-95-1562, Computer Science Department, Stanford University. 138 pages.

A STeP Diagram

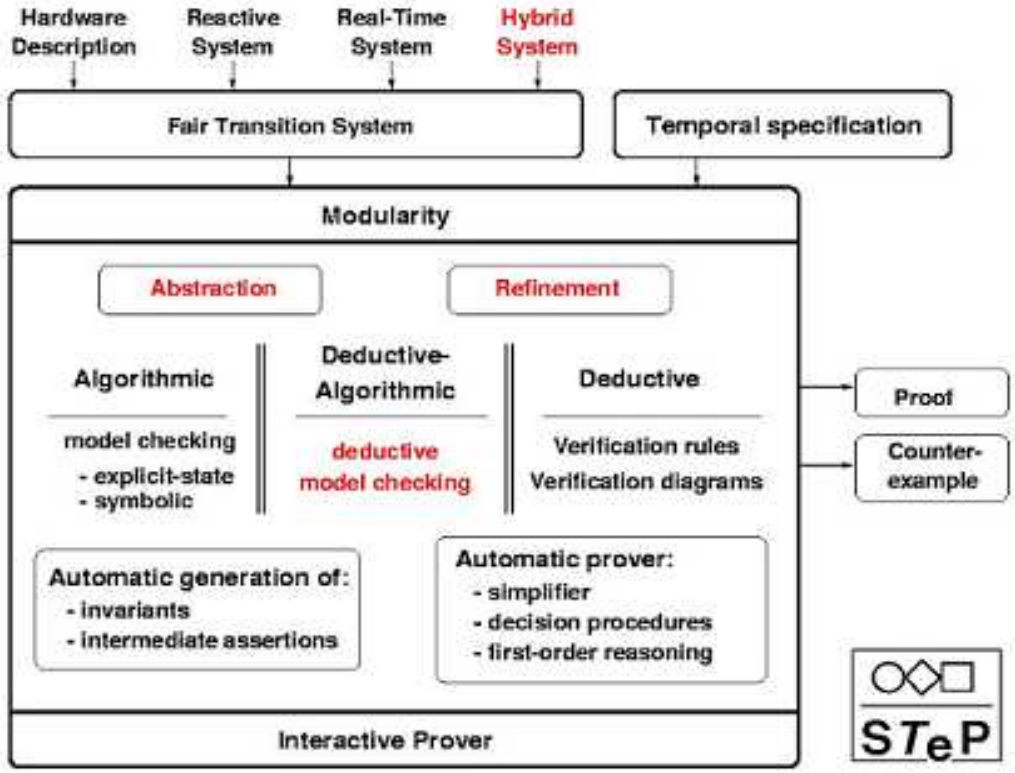

Figure 1: STeP Diagram

B Counterexample for a Specification in STeP

```
The states of the modelchecker are valuations of the following variables:

    B[0] - b
    B[1] - end
    I[0] - pi0

The propositional temporal formula to verify is

    (<>x1)

    where
    ATOMS
        x1 = (B[1]);

    (This formula might differ from the given formula.
    Conjuncts that ensure that is well-formed are added  if necessary.)

INPUT

    SIZES
        BSIZE 2
        ISIZE 1
        CSIZE 0
        FLEX_BSIZE 2
        FLEX_ISIZE 1
        FLEX_CSIZE 0
        RIGID_ISIZE 1
        COMP 0
        JUST 5
        NOFAIR 1

    BNAMELIST "b" "end"
    INAMELIST "pi0"


    HAS_ERR_VAR 0

    MODULE
        INITIAL_COND
```

```
    BOUNDS
        VAR (B'[0])
            VAL false

        VAR (B'[1])
            VAL false

        VAR (I'[0])
            VAL 0

    RELATION true

TRANSITIONS
    TRANSITION "$1_7"
        FAIRNESS_COND: JUST
        TRANSITION_REL:
            DISJUNCT 0
                ENABLING_COND
                    ((I[0]) = 0)
                MODIFYING_REL
                    BOUNDS
                        VAR (I'[0])
                            VAL 1

                        VAR (B'[0])
                            VAL true

                    RELATION true
    TRANSITION "$5_7"
        FAIRNESS_COND: JUST
        TRANSITION_REL:
            DISJUNCT 0
                ENABLING_COND
                    (((I[0]) = 1) /\ (B[0]))
                MODIFYING_REL
                    BOUNDS
                        VAR (I'[0])
                            VAL 2

                    RELATION true
            DISJUNCT 1
```

```
            ENABLING_COND
                (((I[0]) = 1) /\ (~(B[0])))
            MODIFYING_REL
                BOUNDS
                    VAR (I'[0])
                        VAL 3

                RELATION true
TRANSITION "$3_7"
    FAIRNESS_COND: JUST
    TRANSITION_REL:
        DISJUNCT 0
            ENABLING_COND
                ((I[0]) = 2)
            MODIFYING_REL
                BOUNDS
                    VAR (I'[0])
                        VAL 1

                    VAR (B'[0])
                        VAL true

                RELATION true
TRANSITION "$6_7"
    FAIRNESS_COND: JUST
    TRANSITION_REL:
        DISJUNCT 0
            ENABLING_COND
                ((I[0]) = 3)
            MODIFYING_REL
                BOUNDS
                    VAR (I'[0])
                        VAL 4

                    VAR (B'[1])
                        VAL true

                RELATION true
TRANSITION "idle"
    FAIRNESS_COND: NONE
    TRANSITION_REL:
```

```
                    DISJUNCT 0
                        ENABLING_COND
                            true
                        MODIFYING_REL
                            BOUNDS


                            RELATION true
    ATOMS
        x1 = (B[1]);

    PROPERTY
        (<>x1)
    .


The formula does not hold on all the fair computations.
Here is an infinite counterexample:

Path:
-----
    state: {b=F end=F pi0=0} trans: $1_7
    state: {b=T end=F pi0=1} trans: $5_7

Loop:
-----
    state: {b=T end=F pi0=2} trans: $3_7
    state: {b=T end=F pi0=1} trans: $5_7
    state: {b=T end=F pi0=2} trans: $3_7
    state: {b=T end=F pi0=1} trans: $5_7

The strongly connected component from which this loop was extracted is:

state: {b=T end=F pi0=2}
state: {b=T end=F pi0=1}
```